Technical Design Document

Y2 Block A - 2019/2020

Jan Kind
Mathijs Breedveld
Jens Petter

External used libraries

ImGui (link)

This graphical library is used for all visual needs in the engine which are going to be displayed to the user. Apart from displaying objects in the engine, the user will also be able to edit in the engine using the widgets from ImGui.

ImGuizmo (link)

This library which is built to work with ImGui is going to be used to help the user transform objects in the scene view of the engine. This will be done with the help of a “Gizmo” which is essentially what the library ImGuizmo can make.

Rttr (link)

RTTR stands for Run Time Type Reflection. This library is used to inspect and modify objects as runtime. This library is mainly used for the GUI of the engine.

FMOD (link)

FMOD is used for getting audio in the engine which will be used for the game. FMOD supports loading audio files directly but also supports loading banks files which are custom loading files for FMOD.

NLOHMANN (link)

This JSON library is pretty huge. Multiple people in our team are already familiar with this library and since it’s fast and stable to work with, because of these factors we chose to use this JSON library. This JSON library will be used for loading and saving JSON data were loading and saving JSON data can be extended to serialization and deserialization of engine data.

Engine

Serialization / Deserialization

The concepts of serialization and deserialization is the process of making sure that a data type or object can be stored in a format that can also be read. This is needed in the engine, graphics and game in several ways. For example in the engine scenes need to be saved and loaded which also applies to how to undo / redo system will probably work.

Requirements

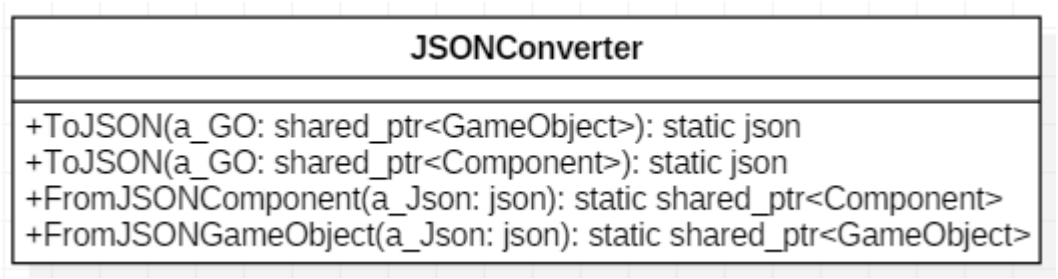
- The engine requires to load and save scenes, there needs to be functionality to implement saving and loading for Components, GameObjects and other custom data that is important to a scene in the engine.
- The serialization and deserialization system needs to be quite dynamic because this implementation might be used by bigger systems like the undo / redo system.

Implementation

- The NLOHMANN JSON library will be used to serialize data to. This library will also be used to read JSON data and convert that JSON data to project data structures and objects.
- RTTR will work with the serialization system as well where all the registered RTTR data from a Component or GameObject will be stored through NLOHMANN to JSON. RTTR will also make sure to get the saved RTTR data back into proper project data structures and objects.
- The undo / redo system will be using the serialization / deserialization system.
- Saving and loading of the engine / saving and loading of individual Components and GameObjects will be using the serialization / deserialization system..

UML

The engine right now only supports serializing and deserializing Components and GameObject since that is what is needed right now, the engine at the moment does not require serialization and deserialization from other engine data structures or objects.



UML of the JSONConverter class that is responsible for serializing and deserializing engine data.

Resource management

A resource management system will take care of managing our resources. This system can and will handle all kinds of resources: models, textures etc. The resource management system will also be displayed and used by the GUI.

Requirements

- This system should be able to load and release resources of any kind.
- This system should be able to load resources from one or multiple directories.
- This system needs to handle the storing of resources IF and only if they are being used in the engine/game.
- When a resource is replaced / renewed then the concept of hot reloading should be activated, also hot reloading of files should be available to the user whenever he or she wants to hot reload certain files.

Implementation

Resources management

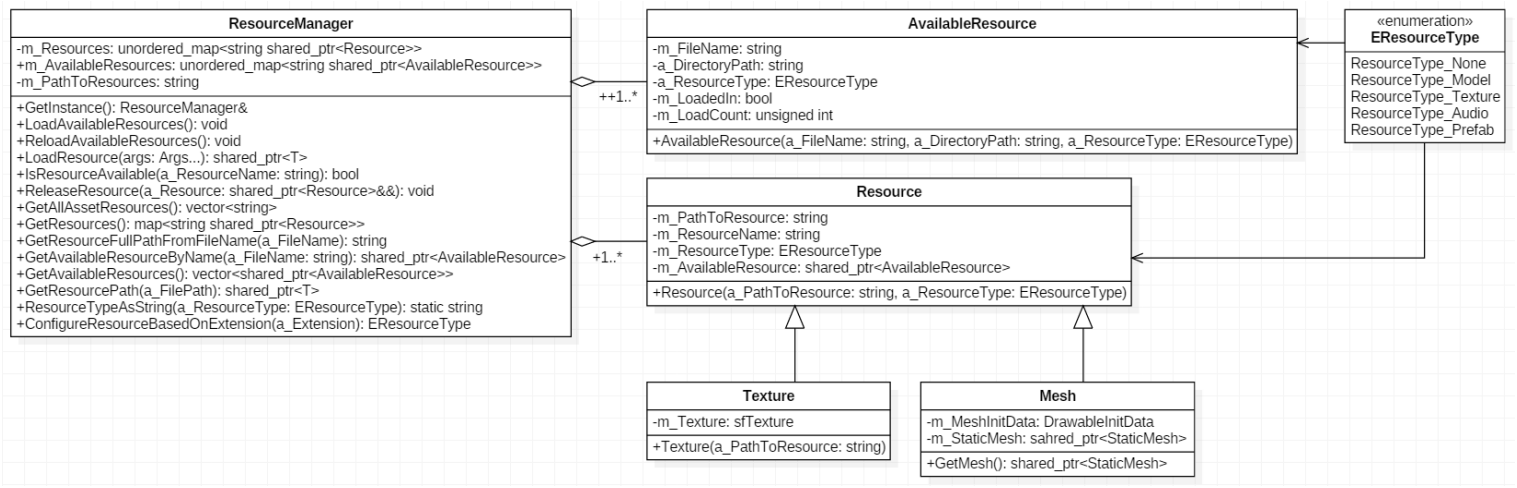
A resource will inherit from a base class “Resource” with all the required resource data. This because resources can range from models to textures but still in the end are resources. This also makes for easy storing of resources where they will be stored in an unordered map with key string and value a shared pointer of this resources. The key will be a known specific resource value which makes it easier to find resources in this unordered map.

When a Component class or any other class needs to access a resource then that class will get the resource from the resource manager. This resource manager will first check if the resource is already loaded or not, if not then the manager creates a resources and returns it, if it is already created then the manager simply returns the already existing resource.

Resource loading

Loading of resources has a twist to it. Every resource that is in a directory “Assets” will be loaded into the engine in the beginning of the game just by name. Because of the fact that only the name is loaded in there is already a clear knowing of which resources can be loaded in if the user chooses to in the engine / game. This already loaded in asset will be a struct or class containing information from the asset like for example the name, asset type (model, texture, etc), is the asset already loaded in etc. The actual assets can then be loaded in based on the already existing loaded in assets. In the end there are 2 asset storage types, the loaded in assets which can be loaded in and the actual assets that can be loaded in by the user.

UML


ResourceManager
-m_Resources: unordered_map<string shared_ptr<Resource>>
+m_AvailableResources: unordered_map<string shared_ptr<AvailableResource>>
-m_PathToResources: string
+GetInstance(): ResourceManager&
+LoadAvailableResources(): void
+ReloadAvailableResources(): void
+LoadResource(args: Args...): shared_ptr<T>
+IsResourceAvailable(a_ResourceName: string): bool
+ReleaseResource(a_Resource: shared_ptr<Resource>&&): void
+GetAllAssetResources(): vector<string>
+GetResources(): map<string shared_ptr<Resource>>
+GetResourceFullPathFromFileName(a_FileName): string
+GetAvailableResourceByName(a_FileName: string): shared_ptr<AvailableResource>
+GetAvailableResources(): vector<shared_ptr<AvailableResource>>
+GetResourcePath(a_FilePath): shared_ptr<T>
+ResourceTypeAsString(a_ResourceType: EResourceType): static string
+ConfigureResourceBasedOnExtension(a_Extension): EResourceType
++1..*
+1..*
AvailableResource
-m_FileName: string
-a_DirectoryPath: string
-a_ResourceType: EResourceType
-m_LoadedIn: bool
-m_LoadCount: unsigned int
+AvailableResource(a_FileName: string, a_DirectoryPath: string, a_ResourceType: EResourceType)
«enumeration»
EResourceType
ResourceType_None
ResourceType_Model
ResourceType_Texture
ResourceType_Audio
ResourceType_Prefab
Resource
-m_PathToResource: string
-m_ResourceName: string
-m_ResourceType: EResourceType
-m_AvailableResource: shared_ptr<AvailableResource>
+Resource(a_PathToResource: string, a_ResourceType: EResourceType)
Texture
-m_Texture: sfTexture
+Texture(a_PathToResource: string)
Mesh
-m_MeshInitData: DrawableInitData
-m_StaticMesh: sahred_ptr<StaticMesh>
+GetMesh(): shared_ptr<StaticMesh>

File system

Since the engine really strictly only connects with an “Asset” directory there is no reason to have too many options for file handling like for example the possibility to open a file search window since the engine focuses on one directory. Extra functionality is however nice to have in the project file system where this system can also be communicated to be used by the engine, game and graphics if set up right.

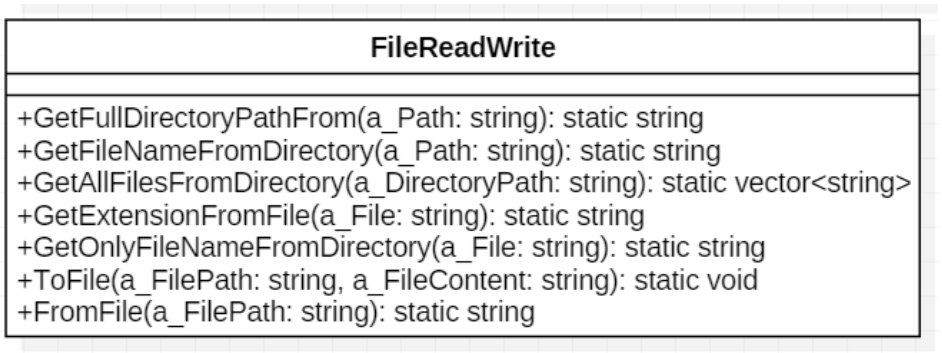
Requirements

- The engine needs to be able to read from a specific directory where files get loaded in.
- The engine needs to be able to read specific values from a loaded in file which can for example be the extension, the full path directory or just the file name.

Implementation

The standard library comes with the Filesystem library in c++ 17. This library is going to be used with our file system. Since the engine restricts communication to disk by only communicating with an “Asset” folder this file system is not going to be huge. The file system is going to be considered to be a helper class with helper functions in most cases. Below is found the UML with very basic static helper functions which are needed in the engine.

UML



Undo / Redo system

The functionality of being able to undo / redo an action in a problem nowadays is quite the standard. Being able to undo and redo your actions are really useful, the engine is going to have this functionality as well.

Requirements

This system really has one requirement that is needed to make this system function which is listed below.

- The undo / redo system needs to be able to when the user makes an action, revert this action and redo the revert as well. The most basic actions that come to mind now are adding / removing Components and GameObjects and changing a Component or GameObject attribute.

Solutions

What I see as one major difference between the possible implementations of a undo / redo system is the way you store events. Events meaning an AddComponent or RemoveGameObject event. I have done my research and came across a few different implementations which I would like to write down here as well. I will be focusing on the different questions on how to store data. I will be talking about the design decisions I have / had trouble with finding the right solution.

Design decision 1: How to store the undo and redo data types?

Solution 1 (storing the undo and redo events as separate data types)

I feel like this is probably the approach most people think of when they think about storing events for a resource management system. For undo and redo functionality there are 2 vectors where one can store new events in and iterate through to find a specific undo or redo event. There is one manager that stores all events but there also 2 iterator integers to keep track of the current active undo and redo event in these 2 stored vectors. Keeping in mind that these 2 vectors don't have to be vectors at all, they count up where a stack can also be used here.

Advantages:

- 2 different storing elements can be hard to manage though you have them separately which can help if one wants to use the undo and redo elements differently because they are separate from each other.

Disadvantages:

- 2 different storing elements (in this case vectors) can end up being hard to manage where these 2 actually very closely work together which suggest for not having these 2 elements be separately stored at all.

Solution 2 (storing the undo and redo events together)

I have not seen this solution yet but after some research I found out that this is also a valid way of dealing with the storage of undo and redo events.

The fact that we have to store events is given to us, we can't go around that fact. One can store a pair where 2 elements can be stored together. That is basically what a pair is: 2 things. We can store lots of pairs where the first element in a pair will be the undo event and the second element in the pair will be the redo event. These pairs can be stored in a vector or stack because really the only thing that will be done is that pairs will be added to the top of the data type. The storage of these pairs come from the fact that all events have a different opposite event, for example the opposite of adding Component is removing Components.

Advantages:

- Storing this data as pairs makes for storing one iterator integer to go over every undo / redo. The undo is the opposite of the redo and the other way around as well. This fact makes storing the undo and redo events together more logical as well.

Disadvantages:

- Storing this data as pairs really tights the undo and redo events together where they can't function without each other. There might be a possibility where undo and redo need to function separate from each other. That is very hard with this setup.

Design decision 2: How to store event specific data?

I need to think about how to store specific event data like for example the attributes of a GameObject when deleting one since one wants to avoid creating specific event holders for every event. My first thought goes to storing any data somewhere in the event which I later can cast to get correct data to use in the redo / undo event. There are some possibilities on storing this data.

Solution 1 (storing event specific data as void* or std::any which is c++17)

I initially did not know what void pointers were and what std::any is. From my understanding is a void pointer a pointer where one can store anything in. Same goes for std::any though std::any is c++ and is type safe. I can store any data here like for example I can store the whole GameObject in memory when I remove a GameObject which I can later use to add the GameObject with the undo system to be back in the editor.

Advantages:

- Void pointer is generally very used in c++, I can see that std::any might be even better since it's c++ 17 and type safe.
- Casting is only needed to go from std::any or a void* to the desired data type, no extra actions required other then casting.

Disadvantages:

- For every event I will end up casting to a specific type, which is not ideal.
- Storing a big GameObject with lots of Components as a copy just for event management purposes can end up being much data.

Solution 2 (storing event specific data as JSON objects)

I can store the event specific data in JSON since the serialization and deserialization functions in the engine use JSON. By storing a JSON I am also not entirely sure right away what type I store but I think I can not go around this fact.

Advantages:

- Using one dynamic system for multiple purposes is I my opinion better than writing hard coded systems for one purpose each only.
- By storing data as JSON objects I simply store one serialized string value which is not that much to store compared to a copy of a whole GameObject for example.

Disadvantages:

- Instead of already having a GameObject or Component in memory I have to first create on from a JSON string. This can end up being heavier and heavier based on how big a GameObject or Component is.

Implementation

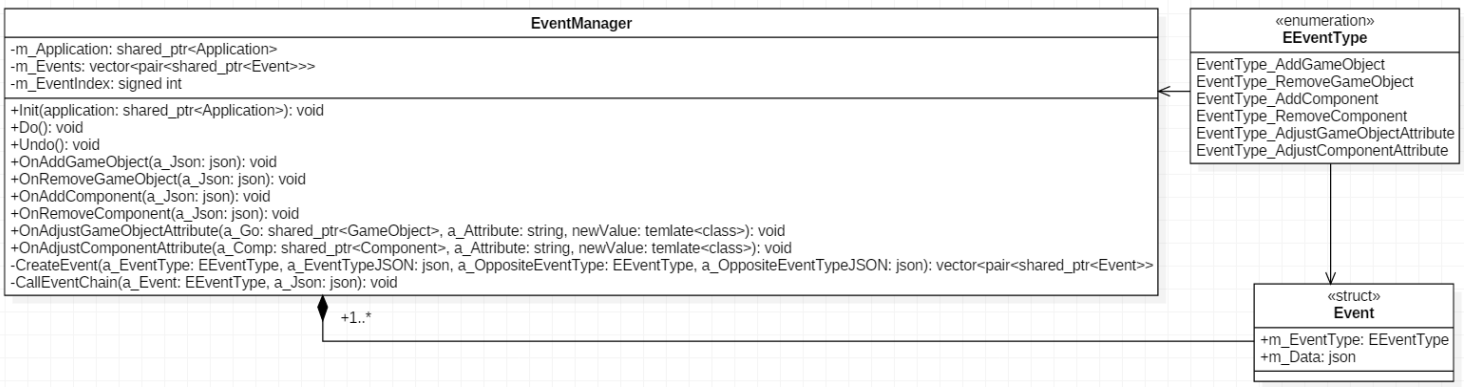
I decided to go with storing the events as pairs as you can see from the UML below here. I see really that every undo has another redo against it so that's why I chose to choose this data type. Now I also have one integer for going through the undo / redo pair to determine which undo or redo event I want to pick.

When for example a GameObject is created in the Entitymanager I want to have that manager call the Eventmanager that a GameObject is created. As you can see from the UML below here there are event call functions for every possible event. Each of these functions will create an event that will get stored but also an opposite event, when a GameObject is create the undo of that event will be removing a GameObject. When one stores GameObjects and Components it will be easier to delete them as well since the undo event will store the GameObject as well and when the removing starts the event will simply remove the GameObject with the same ID as the one that was stored. That is what I would like to achieve here, taking into account that this is for creating / removing GameObject and Components.

I decided that every Event will store JSON data in the form of a serialized GameObject or Component. When a GameObject or Component attribute gets changed I will store the whole object still and override it whenever I want to undo the change I had initially done to the GameObject or Component. I am aware that this is heavier to do then just changing one value though.

When events are created through the functions for each event they will get stored as pairs. Whenever Undo() or Redo() is called I will run different functionality based on the event data I get from the event index integer I store that corresponds to an element in the data object with pairs. Control z will call Undo() and control y will call Redo()

UML



GUI Creation and management

There are a specific set of elements needed in this engine for example a way to view every GameObject in a current engine view, a way to add / remove components, a way to load in files such as render files / font files etc. These functionalities all require a (or more) window(s). This asks for a clever way to create GUI windows since there are a lot of windows that needs to be created, below is listed the possibilities which were considering while thinking about the setup of this system.

I think for creation of the GUI there are 2 possibilities which are listed below.

Solution 1 (every window in one class)

Having all windows in one class. I think this solution speaks for itself. Just hearing the title of this solution makes me not want to use this for creation of the GUI but I still would like to consider the advantages and disadvantages of this solution.

Advantages:

- Less code.
- Everything is nicely packed in one place, creation can even be seperated in other functions.

Disadvantages:

- A very very very large file containing all the GUI windows.
- Possibly complicated overview of what is going on where.

Solution 2 (every window is one class stored inside a manager)

This solution also speaks for itself I think, every window as a GUI window is a separate class where only the functionality of that window is done in the binded class. Every custom window class will inherit from a base window class where in the manager will be a vector stored of these window classes.

Advantages:

- A clear overview of a specific GUI window.
- With the help of a base class for each window we can store a vector of base window objects inside the manager, no pointer to each custom window which makes for a lot of pointers in the manager class.

Disadvantages:

- More code.
- Getting a custom window would require the manager to loop over every custom window, this also applies to modifying a custom window object.

Conclusion

Even though I didn't like solution 1 from the beginning, I still wanted to consider its advantages and disadvantages. After writing these down I am becoming more and more fan of solution 2. This is the solution that I will be going for when it comes to creating GUI.

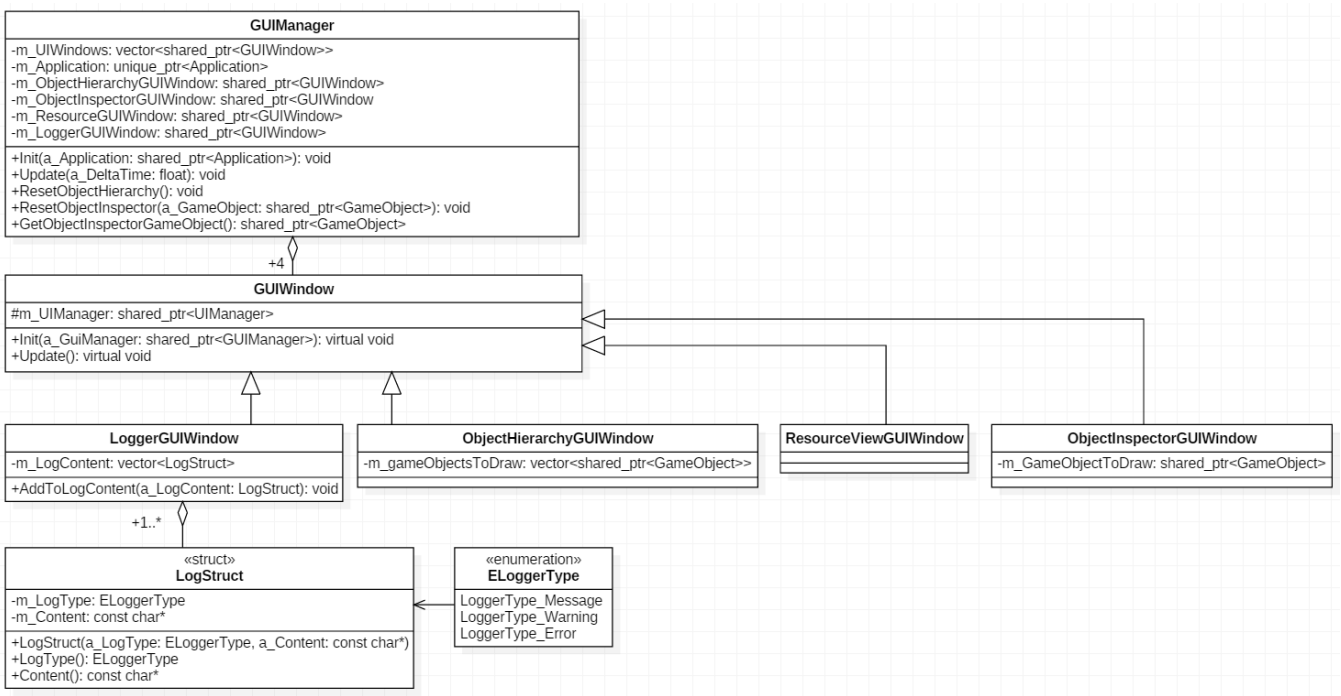
Implementation

Like described in solution 2, every GUI window will be a base GUIWindow class. All these GUIWindow classes will be stored inside a manager and updated through this manager as well. Individual access towards one specific GUIWindow will be done through the manager as a getter function. An Init and Update function will be provided to every custom GUIWindow class.

The Init function will initialize the custom GUIWindow and the Update function will make sure to draw the GUIWindow if needed and also will handle ImGui events on that specific GUIWindow such as clicking on a button if there is a button or clicking on an image if there is an image etc. The corresponding function that gets called through these events will be separated into their own functions in the custom GUIWindow class.

Below is a UML found where is described how the management and creation of GUI Windows is done.

UML



Communication between GUI WIndows

The user (gameplay programmer) is eventually going to make components themselves. Making sure that the user has to as little as possible to make components pop up in the gui.

The option to choose to make a component inspectable in the inspector is something that would be nice to have, this would mean that there would be components that can be not visible in the editor but still can run. This requires a boolean of some sort to make sure to set or not set the component to be inspectable in the gui.

The user needs to always make sure to register a component through rttr registration calls, there is no way around this. The user also inherits a custom component from the base class component and overwrites certain methods. The fact that this is set in stone throws away a lot different implementation methods but also makes it easier to make sure the data rendered on screen since you can get all kinds of data with RTTR.

Implementation

The user needs to register RTTR types and make their component inherit from the base class Component. When a custom component is registered and added to a GameObject it will automatically be shown on screen. This makes the user not having to deal with GUI stuff which is very ideal in this case I think.

The custom windows for making sure that components draw on screen deal with the communication to the desired manager classes to make sure to draw entity related or other related stuff to the screen through the GUIManager.

The through the GUIManager part is very important because the engine is going to have a hierarchy from important to less important classes. A GUIWindow class for example should never ever access another less important class like a GameObject directly. This is going to be done through the manager, through the Application class and down downwards again to the EnityManager so that it will land in the GameObject class. This way the engine displays a very structured and neat class hierarchy.

Logging

Logging in the concept of visually displaying messages specified by the user or the engine towards the user when certain events happen. This system is going to be installed in the engine as well.

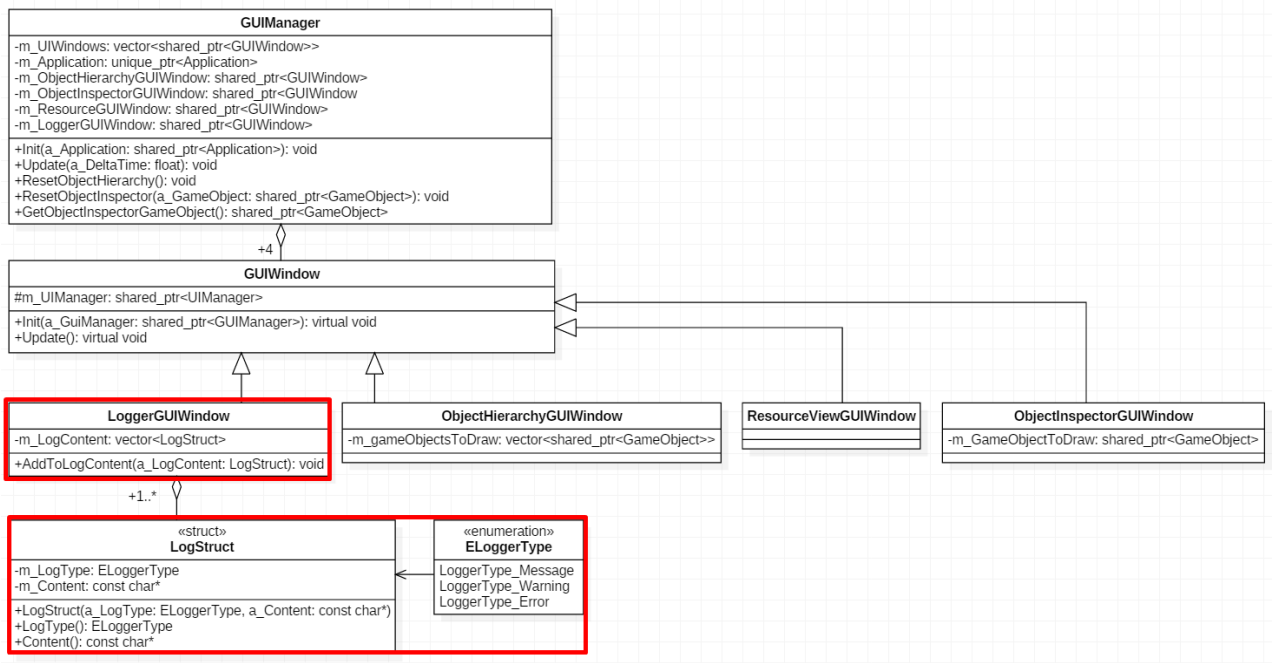
Implementation

Logging in the engine will be implemented in the engine while looking at how Unity's logging system works visually. Unity has a GUI window where logging messages are displayed whenever the user decides to log in their own scripts. Messages can be displayed in colors, as far as I know they can be displayed in gray for a normal message, yellow for a warning and red for an error. An error message will also stop the game from running.

The logger in our engine will be behaving in the same way but with a twist. Optionally the user will be able to display engine related events to the logger as well such as the adding of an object or clicking on a window etc. This option will visually be available to the user in the logging window.

Below is an overview of the UML shown that corresponds with how the Logger system will be implemented.

UML



ECS

*A*n ECS (Entity Component System) is going to be used for the core architecture of objects in both the game but also the engine as well. An ECS where Entities are objects or keys to be stored, Components are data objects and Systems are systems that manipulate these Components. Important is (and what I got mixed up while doing research to this in the beginning as well) is that entities and systems don't know anything about each other.

It is also allowed to make an EC system (Entity Component system) for this project where systems are out of the question here and components are only used. Components operate on other components directly where with an ECS systems try to operate on all components of the same type at once, this is not the case with an EC system. This implementation is much less complicated because systems are not part of this implementation, everything will be component based.

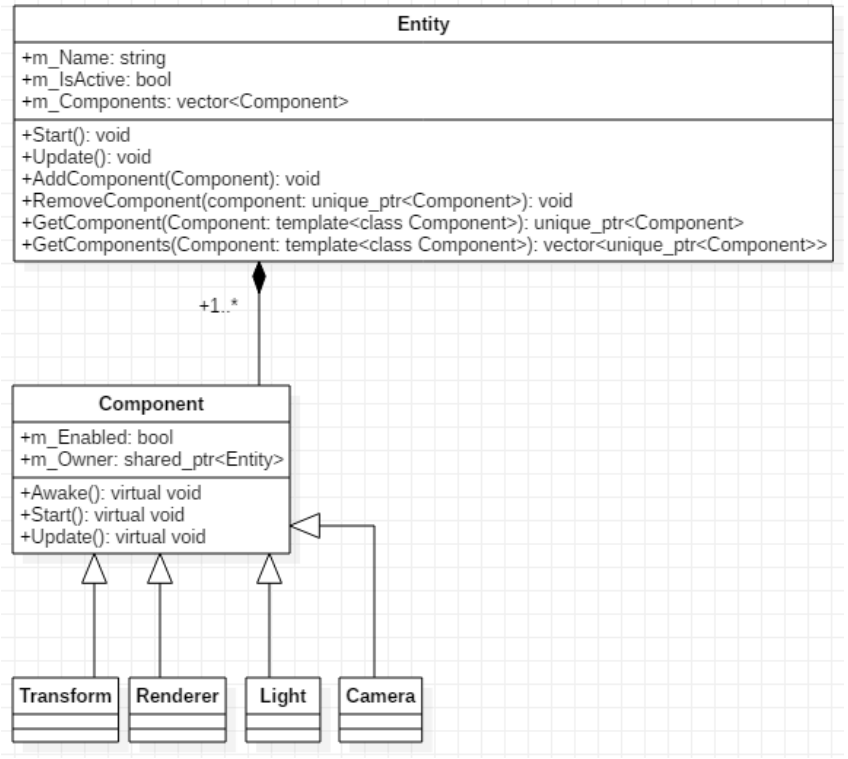
Though EC systems sound a more friendly approach for implementation then an ECS. I will research both EC and ECS implementation because I would like to know what is possible.

Requirements

- An ECS system needs to be created, though the implementation is free to not use systems. If decided for an implementation without systems that will be fine as well.
- Considering an ECS. Entities can be keys/ID's or very simple objects components are essentially data and systems are the systems that need to manipulate the data in the Components. The clear separation between entities, components and system needs to be really there.
- The storage of the entities, components and systems comes when considering an ECS implementation comes down to the implementation of ECS. There are a number of ways to implement ECS where I will take several methods into account before choosing the final implementation I will go for.
- Decisions need to be made on how limited components can be, do components really only store data or is a simple function such as GetPosition on a Transform component also allowed, deciding this will also change how much systems need to do if ECS is chosen as an implementation method.

Solution 1 (EC implementation)

This system is what I like to see as a more Unity way of how Unity used to display components and entities to the user.but with my own twist. What happens here is that entities and components are only used in this implementation. Entities own a bunch of components and can own custom components made by the user. Entities update components when needed to, for example an PhysicsBody components need to update so that the entity that uses this component falls down like expected of a PhysicsBody.



UML of solution 1

Advantages:

- Very simple to understand interface, there are only entities and components where ethe entity makes sure components can update.
- Time wise this implementation is the best to consider because it is as I mentioned above to my opinion the simplest implementation out there.
- A basic understanding of template programming is needed.

Disadvantages:

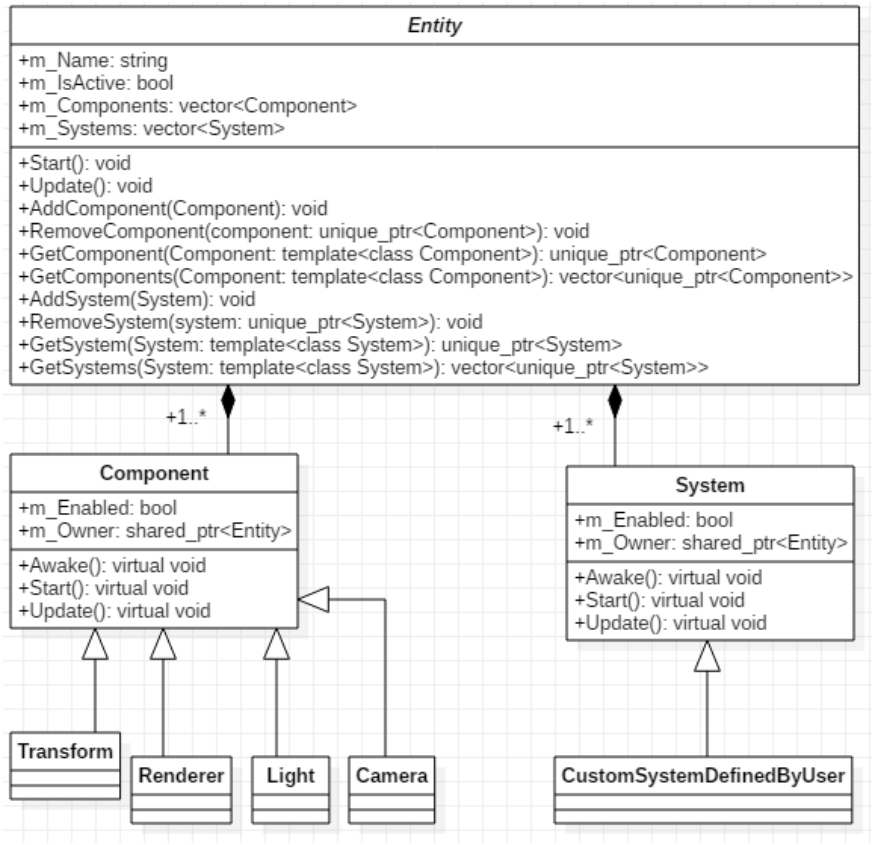
- LOOPS. To get a component you need to loop through all components, this can of course be optimized by something like using an index or can be fixed by predefining components in a static place but this can only be optimized like this to a certain extent.
- Components are stored inside entities, this gives for an extra layer of looping when needed to get all components if needed at some point.

References/links:

- Overview of the old Unity entity component system.

Solution 2 (EC system with custom components (systems) defined by the user)

Entities hold and keep track of Components and Systems. This means that when an Entity exists or is constructed a component or system can be directly added to the Entity. Adding will not take any tours to other classes, adding will go directly through to the entity. What systems here is basically modify the data in components, which essentially systems can also be called custom components created by the user. Though there is a clear separation between Components and systems (custom components). This implementation calls for the user defining custom systems, not components. The components are defined by the engine.



UML of solution 2

Advantages:

- A basic understanding of template programming is needed.
- A clear overview to the user with only components.

Disadvantages:

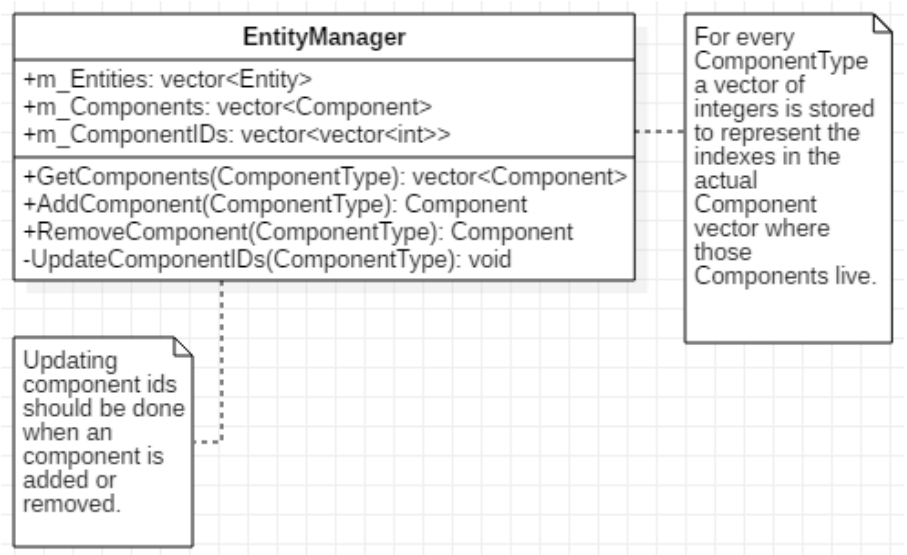
- Components and systems are stored inside the actual entity. Having many entities that needs to loop through their components and systems each update can be quite a performance mess I think. The optional fix for this is to store every component and system somewhere else which requires another implementation.

References/links:

- Entity Component Systems lecture by Bojan (04/09/2019)
- Blast (Bojan his project which he showed in the Entity Component system lecture) (04/09/2019).
- Overview of the new Unity ECS system.

Solution 3 (EC system stored in a manager class)

In this solution entities, components and optionally systems are stored in some kind of god class that keeps track of entities, components and optionally systems. Most of the time it will probably be some sort of manager class.Entities, components and optionally systems will be stored in this manager class in separate vectors or other data types that are preferred to store such objects. This whole idea of this approach is that every single data type talked about in these approaches is stored together somewhere. Which gives for easier access to entities, components and optionally systems then other approaches.



UML of solution 3 with only an EC implementation in mind.

Advantages:

- Everything is stored in one place, which gives a clear overview of everything that is stored.
- Though everything is created in a manager, entities still have a vector of component pointers that point to the actual components in the manager. This way old GetComponent functionality is also supported.
- When later chosen to implement a scenemanager, entities can easily be set to not destroy when a scene is ended which gives for an entity that is essentially a “singleton entity”.

Disadvantages:

- Time wise this implementation takes more time in general also to set up, the code base is larger here then for example the other solutions.
- When a scenemanager is introduced, the resetting of entities and components needs to be done which requires a system for this to happen.
- When using an ECS approach, systems needs to access all the components with one type which can be done of course.

References/links:

- A 3,5 hour youtube playlist of someone who worked on the sims who explains how this system works and can be implemented (Link to the whole playlist here).
- GDC talk by Timothy Ford, the lead gameplay programmer of overwatch where he shows their ECS for overwatch (Link to the GDC talk on youtube here).
- Entity Component Systems lecture by Bojan (04/09/2019)
- Datastructures and Algorithms lecture by Bojan (04/09/2019)

Conclusion

There are a lot of different ways to implement either ECS or EC. I think I am going with an EC system just to make my life easier. I see the benefit of an ECS but managing and making a really tight good one is harder to do then having an EC. Also I see very many similarities between the old Unity EC system which I like because looking at Unity which is familiar to me helps me understand how I would like to build systems.

Now that the decision is made that an EC system is made, I would also like to say that I would like to go for the system where my entities and components are saved in a manager. This gives many benefits like for example that everything is stored in one place, I can choose to update components through this manager instead of the entities or I can with a simple implementation get all entities of one type. I see many benefits in this approach which is why this approach is the one I will go for. The only downside of this system is that I did not build this before so there might be problems I will walk my head against but I have a pretty clear overview of how I would like to implement it so I think I should be fine.

Implementation

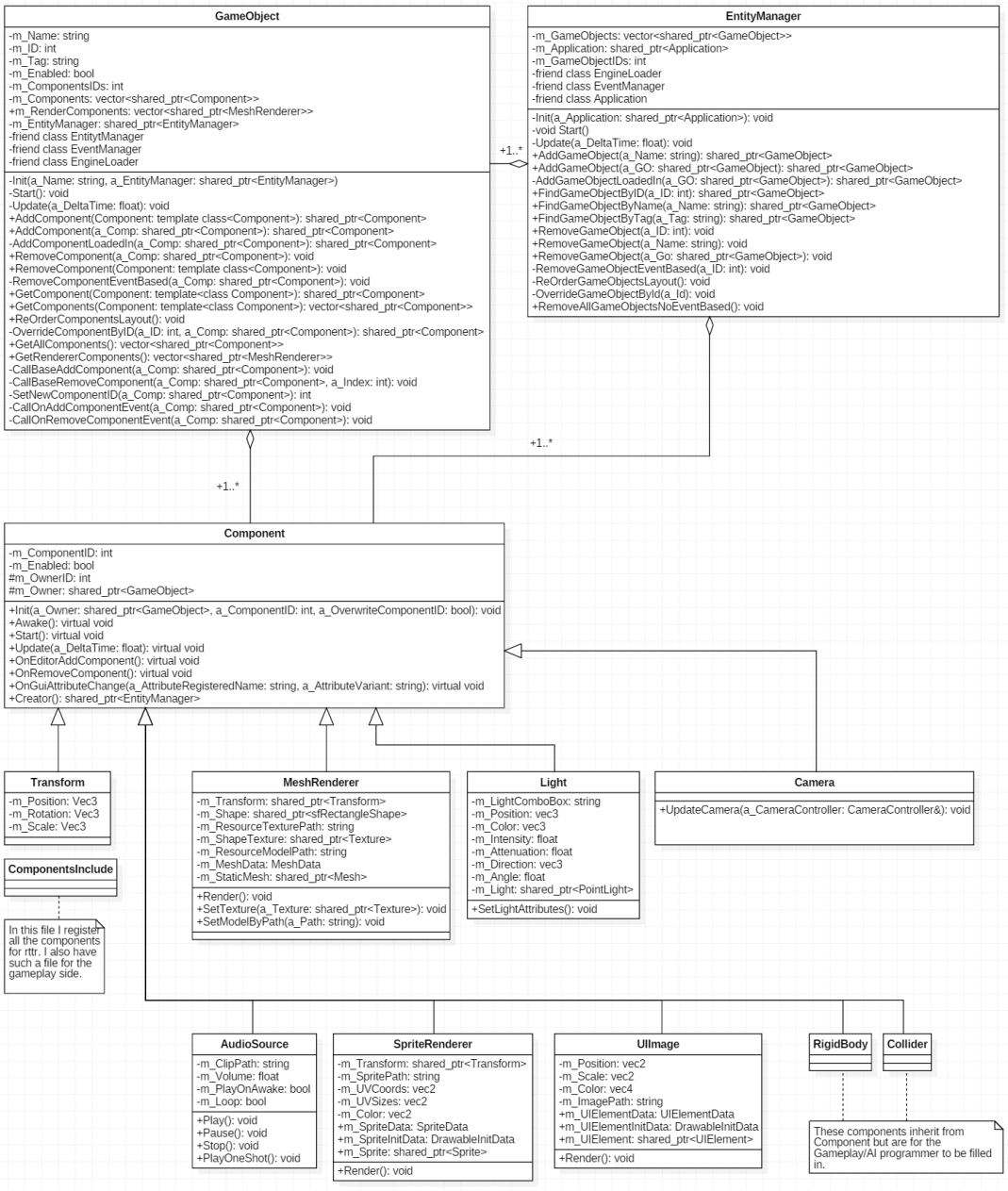
I would really like to challenge myself which I think becomes hard considering all the functionality the Engine needs to have. That's why I decided to build a manager which is a bit harder than solution 1 but it is still something that should be up and running quite fast.

How I would like to do this is to store an integer vector for each component. This integer vector gives me back indexes in where the component lives in the overall vector with all components. Example: A Transform component can live in index 0, 4, 13 for example of a vector of components of size 20 maybe. This index vector will update when a component of a type is added or removed because then of course the vector with indexes may change.

Another way to work with this instead of indexes is a vector inside of a vector where each vector in a vector contains the same components. So each vector inside that first vector belongs to another type of components. For this method one only needs to store the index of where for example the Transform components can be found. Which can be done as a static value inside that component itself.

These are 2 different ways of tackling this problem IF this system is needed to be implemented later on in development if a special feature is needed. These 2 methods will be time checked where the method that takes the least amount of cycles will be used.

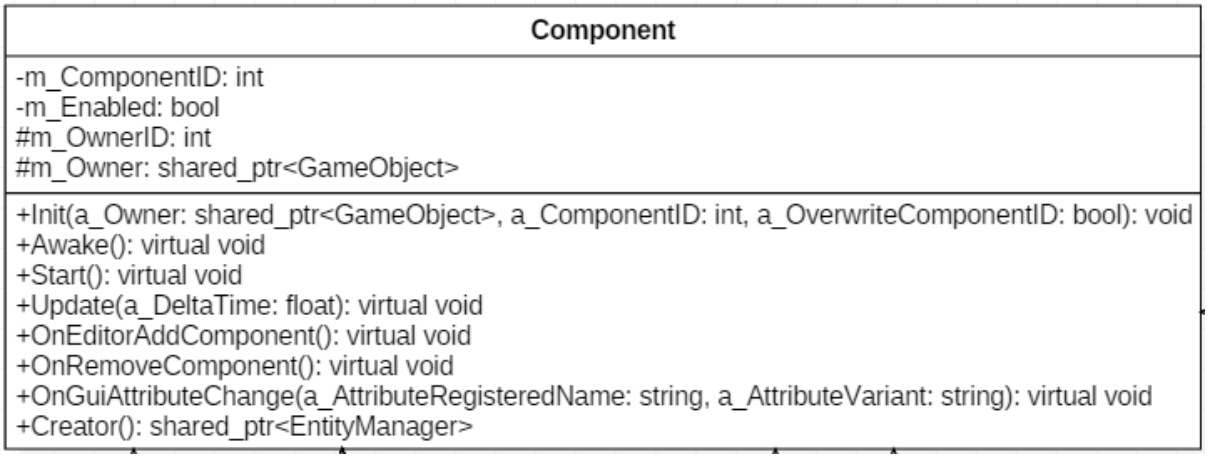
Below you will see the UML of the system that will be implemented.



UML of the final implementation solution for the EC system that can easily be expendable to an ECS system

Components

Components inherit from a Component class, this makes for easy storage, less writing of code and cleaner code in general.



UML of the Component class

Transform

The transform component is of course to keep track and manipulate the position, scale and rotation of an object. One must not forget that this is the transform for an GameObject where if a mesh is introduced as a component as well, will also be the transform of that mesh because the Renderer is also a component. Since it is almost 100% certain that every GameObject will have a transform, the system will make sure that every Gameobject has a Transform component by default.

Renderer

The engine will have a separate class that handles all communication with the graphics API so that no random calls to the graphics API are all scattered across the engine code. The There will be 3 components regarding graphics that will be in the editor: a MeshRenderer, SpriteRenderer and UIImage component.

MeshRenderer

The mesh renderer component draws depending a sfml texture or a gltf model depending on the graphics target that is currently being used in the engine. Additional functions to change the model etc will also be provided to the user of the engine through this component. The transform component will be responsible for the position, scale and rotation no matter which graphics target is used for the engine.

SpriteRenderer

This component will only work when using the graphics target for the engine that uses directx to render objects. Internally this component will behave very similar to the MeshRenderer component, the difference with this component is that it draws sprites instead of meshes.

UIImage

The UIImage component draws a UI element of course. This component will only work when using the graphics target for the engine that uses directx to render objects. The position and scale of this element need to be set in the component itself since the values of position and scale range from a number from 0 to 1. This is a requirement from the graphics target to the engine but this way multiple UIImage components can exist on one GameObject which makes for the fact that the user can have one GameObject be responsible for the whole UI. This is also handy because the engine doesn't support children of objects.

Light

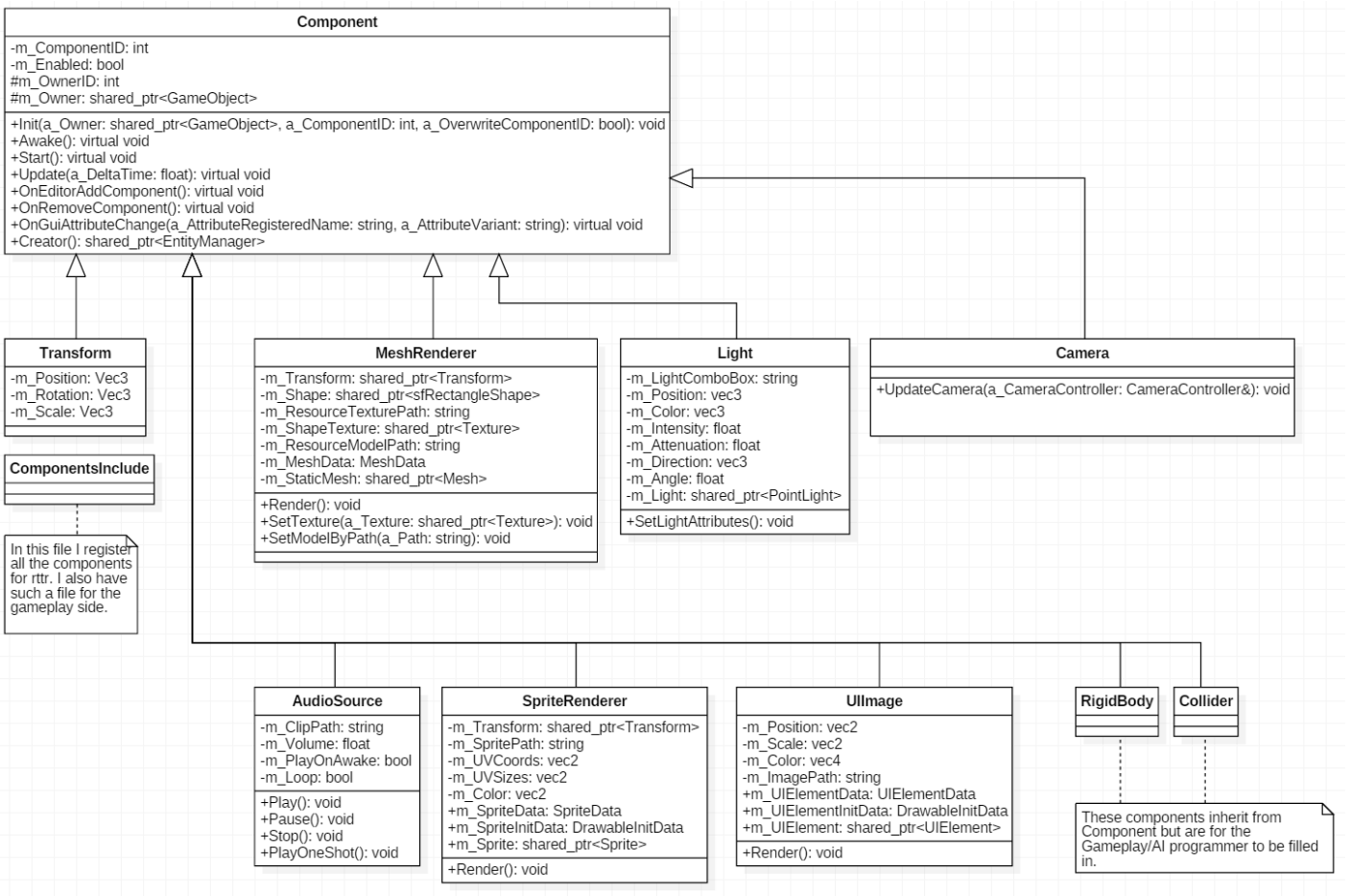
A Light component is a bit different then all the other components behind the scenes. A light is made on the graphics API side, where a shared ptr will make sure the engine can also access that light. Destroying and creation of light is also done in the graphics API side and can be called for with the custom class on the engine side where you can only call graphics API calls from. Spot, point and directional lights are going to be implemented where different variables for the light is going to be used based on the light type itself.

Camera

A Camera is created on the graphics side, where with the help of a helper function the camera is taken and saved from the graphics side. The transform of the GameObject where this Camera component lives on will determine the position, rotation and scale of the camera.

Implementation

Below will be shown a UML diagram of the components and its variables.


Component
-m_ComponentID: int
-m_Enabled: bool
#m_OwnerID: int
#m_Owner: shared_ptr<GameObject>
+Init(a_Owner: shared_ptr<GameObject>, a_ComponentID: int, a_OverwriteComponentID: bool): void
+Awake(): virtual void
+Start(): virtual void
+Update(a_DeltaTime: float): virtual void
+OnEditorAddComponent(): virtual void
+OnRemoveComponent(): virtual void
+OnGuiAttributeChange(a_AttributeRegisteredName: string, a_AttributeVariant: string): virtual void
+Creator(): shared_ptr<EntityManager>
Transform
-m_Position: Vec3
-m_Rotation: Vec3
-m_Scale: Vec3
MeshRenderer
-m_Transform: shared_ptr<Transform>
-m_Shape: shared_ptr<sfRectangleShape>
-m_ResourceTexturePath: string
-m_ShapeTexture: shared_ptr<Texture>
-m_ResourceModelPath: string
-m_MeshData: MeshData
-m_StaticMesh: shared_ptr<Mesh>
+Render(): void
+SetTexture(a_Texture: shared_ptr<Texture>): void
+SetModelByPath(a_Path: string): void
Light
-m_LightComboBox: string
-m_Position: vec3
-m_Color: vec3
-m_Intensity: float
-m_Attenuation: float
-m_Direction: vec3
-m_Angle: float
-m_Light: shared_ptr<PointLight>
+SetLightAttributes(): void
Camera
+UpdateCamera(a_CameraController: CameraController&): void
ComponentsInclude
In this file I register all the components for rttr. I also have such a file for the gameplay side.
AudioSource
-m_ClipPath: string
-m_Volume: float
-m_PlayOnAwake: bool
-m_Loop: bool
+Play(): void
+Pause(): void
+Stop(): void
+PlayOneShot(): void
SpriteRenderer
-m_Transform: shared_ptr<Transform>
-m_SpritePath: string
-m_UVCoords: vec2
-m_UVSizes: vec2
-m_Color: vec2
+m_SpriteData: SpriteData
+m_SpriteInitData: DrawableInitData
+m_Sprite: shared_ptr<Sprite>
+Render(): void
UIImage
-m_Position: vec2
-m_Scale: vec2
-m_Color: vec4
-m_ImagePath: string
+m_UIElementData: UIElementData
+m_UIElementInitData: DrawableInitData
+m_UIElement: shared_ptr<UIElement>
+Render(): void
RigidBody
Collider
These components inherit from Component but are for the Gameplay/AI programmer to be filled in.

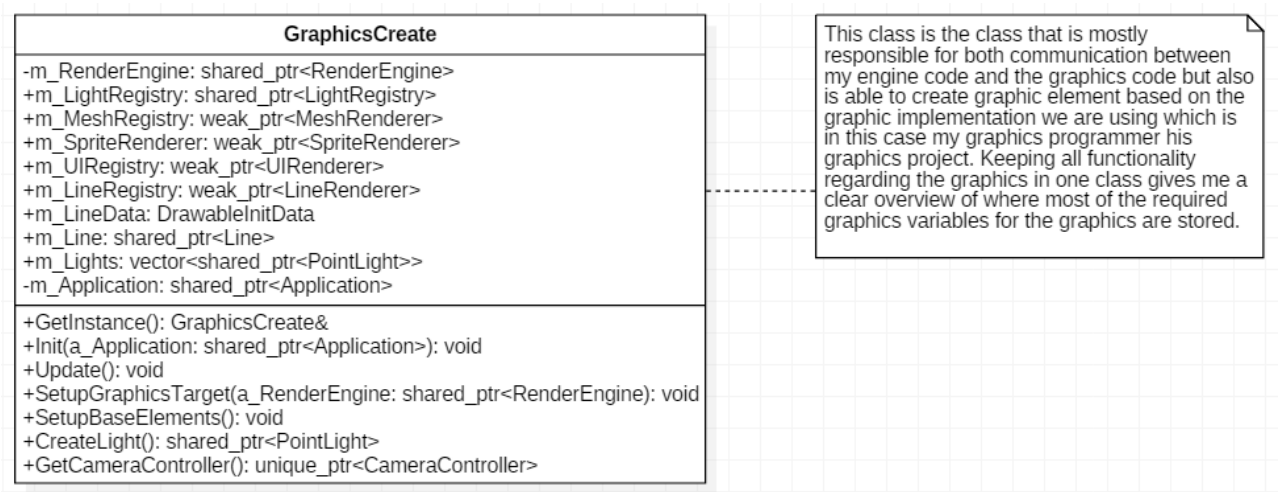
Communication between the engine and the Graphics Engine

Ideally the communication between these 2 project would be minimum where with minimum requirement the graphics engine can be taken away from the engine and another graphics engine can be easily installed in the engine project. Also calls from all over the place in the engine project to the graphics engine project is not really that clean. When creating this system there should be also taken into account that the graphics engine can grow over time where more graphic elements might be able to be created over time where the engine should be able to quickly implement this new feature on the graphics engine as well.

Implementation

All the facts above made me decide to make sure most of the calls that want to access the Graphics Engine project are called to a class that is still inside the engine. This class however is the only class that really communicates with the real graphics engine in the engine project. All functionality with the graphics engine project is done in this class apart from the window calls which are done in the window management of the engine. The aim for this communication class was to have one class be responsible for functionality between the actual engine and the render engine which I think this class does really well.

UML



Gameplay

AI decision making

AI needs to be able to choose it's behavior during gameplay.

Requirements

- *AI needs to be able to switch to different states*
- *AI needs to be able to change patterns based on certain factors (Like when it takes enough damage)*

Solution(s)

FSM

A Finite State Machine uses states to define the behavior of the entity. The entity can only be in one state at once.

Advantages:

- *Simple solution to implement and create*

Disadvantages:

- *The more states you have to more complex the FSM gets due to the amount of connections the state machine will get. This can be fixed however by implementing Pushdown automata to the system.*

References/links:

- https://www.youtube.com/watch?v=hJIST1cEf6A
- https://gameprogrammingpatterns.com/state.html

GOAP

GOAP is a method were you give an AI a goal and it would work out how to get to that goal based on the actions it could do.

Advantages:

- *Allows for more complex AI*
- *Allows the state machine to be a lot more dynamic*

Disadvantages:

- *Requires more time to implement*
- *Is a huge overkill for our project*

References/links:

- https://gamedevelopment.tutsplus.com/tutorials/goal-oriented-action-planning-for-a-smarter-ai--cms-20793

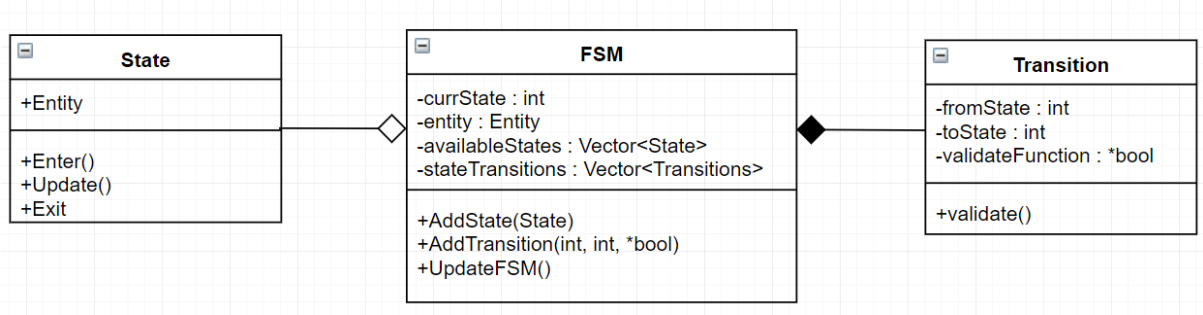
Conclusion

Since our project isn't that large and the AI isn't that advanced the right choice would be to choose Finite State Machines. They allow us to make the AI decision making in a faster time.

Implementation

Interface:

This system will work by making the user be able to create states and the transitions between them. It does this by creating state child classes and entering them into the FSM then they create transitions between them by creating transition object.



Physics engine

Engines

For R-type delta we need a bunch of physics to be done while playing.

Requirements

- *Collision to be done between objects*
- *Events being sent when a collision happened.*
- *Objects need to be able to be affected by gravity.*

Solution(s)

Box2d

Box2d is a physics engine used for 2d objects. This will do everything a physics engine is supposed to do right out of the box.

Advantages:

- *It is already done and easy to implement.*
- *Already works with IMGui*
- *Is well documented*

Disadvantages:

- *Does things we will not need in the project*
- *Could be slower at doing things then if we did them ourselves*
- *It is only for 2d physics. So doing things with the background will be harder to do. This can be done by putting things in the background at a different layer for example.*

References/links:

- https://box2d.org/about/
- http://blog.teamthinklabs.com/index.php/2011/11/30/pseudo-3d-collision-detection/

ReactPhysics3D

React Physics 3d is another physics engine that could be used for the project

Advantages:

- *It's for 3d objects so we have access to 3d collision*
- *Has mesh collision*

Disadvantages:

- *It will be overkill.*
- *There is only documentation for it no videos etc.*

- *Does not have triggers*

References/links:

- Websites, articles, any research to confirm the advantages and disadvantages. Any link to what you're talking about. Libraries.

Bullet Physics

Bullet physics is a real-time collision detection and multi-physics simulation for VR, games, visual effects, robotics, machine learning, etc.

Advantages:

- *Good quality physics*
- *Has everything we need for 3d physics and more*

Disadvantages:

- *Might be overkill since were only going to be using a small portion of it*

Making a physics engine ourselves

Since the things we need for physics aren't that much we can also just make the things we need ourselves

Advantages:

- *Will do exactly what we need*
- *Very flexible*

Disadvantages:

- *Will require more time to build and research*

Conclusion

For this project, I prefer to make the physics engine ourselves because this is the way to learn the most and since this game doesn't require a lot of things we can do it with less time. If we do have too little time to do it we can resort to using Box2D since for the things we need to do we do not need 3d collision

Collision system

The physics engine needs a way to check for collisions to send back information.

Requirements

- *Be able to do a few collision types*
    - *AABB*
    - *OBB*
    - *Sphere*
    - *Ray*
- *Be able to have static and moving collision*
- *Objects need to be able to subscribe to collision events*
    - *Collision enter, exit*
    - *Trigger enter, exit*
- *Send information like hitpoint, hit normal, etc.*
- *Be able to set layers*

Solution(s) Broad-Phase

Implicit grid

Advantages:

- *Really fast.*
- *It doesn't take up a lot of memory for small object pools.*
- *Good for objects with varying sizes.*

Disadvantages:

- *Isn't good when there are more then a 1000 objects on the screen*
- *It is only a fixed size.*

References/links:

- Intro to Game Physics
- Real-Time Collision detection page 291

Tree

Advantages:

- *Can handle large levels of islands*
- *Good for objects of varying sizes*

Disadvantages:

- *Have to worry about balancing the islands.*
- *Often slower then grid solutions*
- *Harder to write and use*

References/links:

- Intro to Game Physics
- Real-Time Collision detection page 241

Sweep and prune

Advantages:

- *Infinite bounds*
- *Low memory consumption*
- *Easy to understand and write*

Disadvantages:

- *Can be slow*
- *Can produce false positives*
- *Testing more than one axis is non-trivial*

References/links:

- Intro to Game Physics
- Real-Time Collision detection page 241

Conclusion

For broad-phase collision, it is best that we use the Implicit grid because we only have to do collision for things that are visible on the screen others we can just ignore. Our objects also don't over 1000 objects.

Implementation

Interface:

This system will work by giving the system a list of colliders and then run the system.

Further details:

Links to other documents. List of files or classes involved.

Graphics Engine

Shading Model

In the original R-Type Delta, a lot of light effects are pre-baked into the animated textures. When an enemy is hit by a light-emitting object, the whole texture becomes brighter. This was a clever way of doing lighting at the time while also indicating damage being done, but we are targeting much more powerful systems. This allows us to do many actual lighting calculations. In the game, there are possibly hundreds of lights in the scene at the same time. No shadows are in the game.

Requirements

- *Lighting needs to be calculated for each light in the scene.*
- *Reduce the need for pre-baked light animations which would require clever artists.*

Solution(s)

Forward Shading

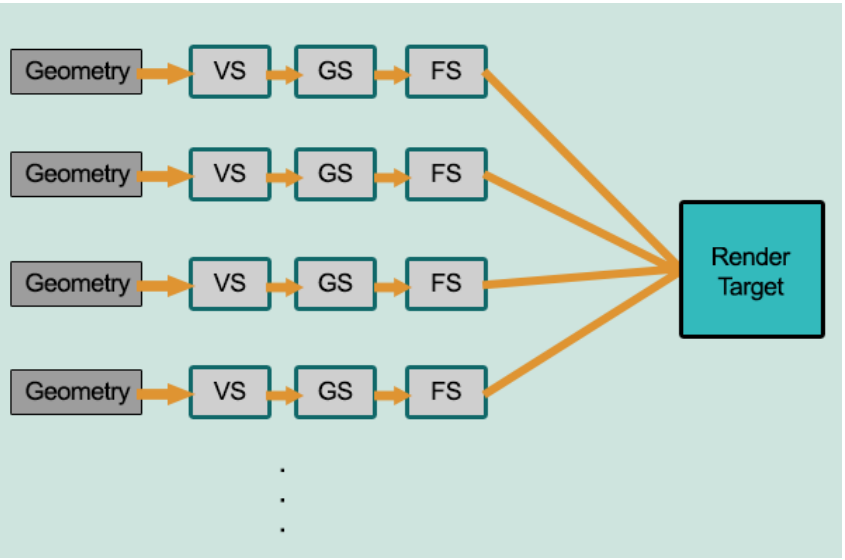
In forward shading, lighting calculations are done as geometry is being drawn in the fragment shader.

Advantages:

- *This is the most straightforward way to render. It is easy, and usually the default way to handle things.*

Disadvantages:

- *The problem is that even geometry that is not visible will have to do light calculations. This is quite expensive.*



Deferred Shading

Deferred shading means that all geometry is first drawn to a buffer, storing the pixel color from a texture, the normal at that pixel and the depth of that pixel. Once this buffer is filled, the light is calculated for each pixel in this buffer.

Advantages:

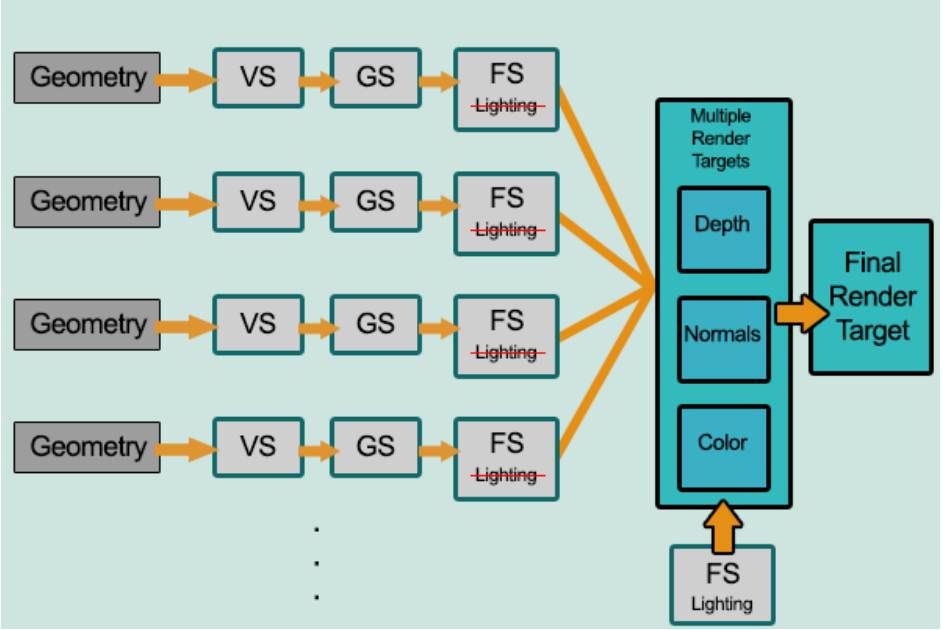
- *This way, no light calculations are required for pixels that are not finally visible on the screen, thus saving time.*

Disadvantages:

- *The main disadvantage is that transparent objects become hard to render. They can be drawn separately, but this requires some workarounds to get the final pixel color correct.*
- *Multiple render passes are required to get the final image on the screen. This means it is more complicated to write and maintain.*

References/links:

- https://learnopengl.com/Advanced-Lighting/Deferred-Shading
- https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342
- https://martindevans.me/game-development/2015/10/09/Deferred-Transparency/



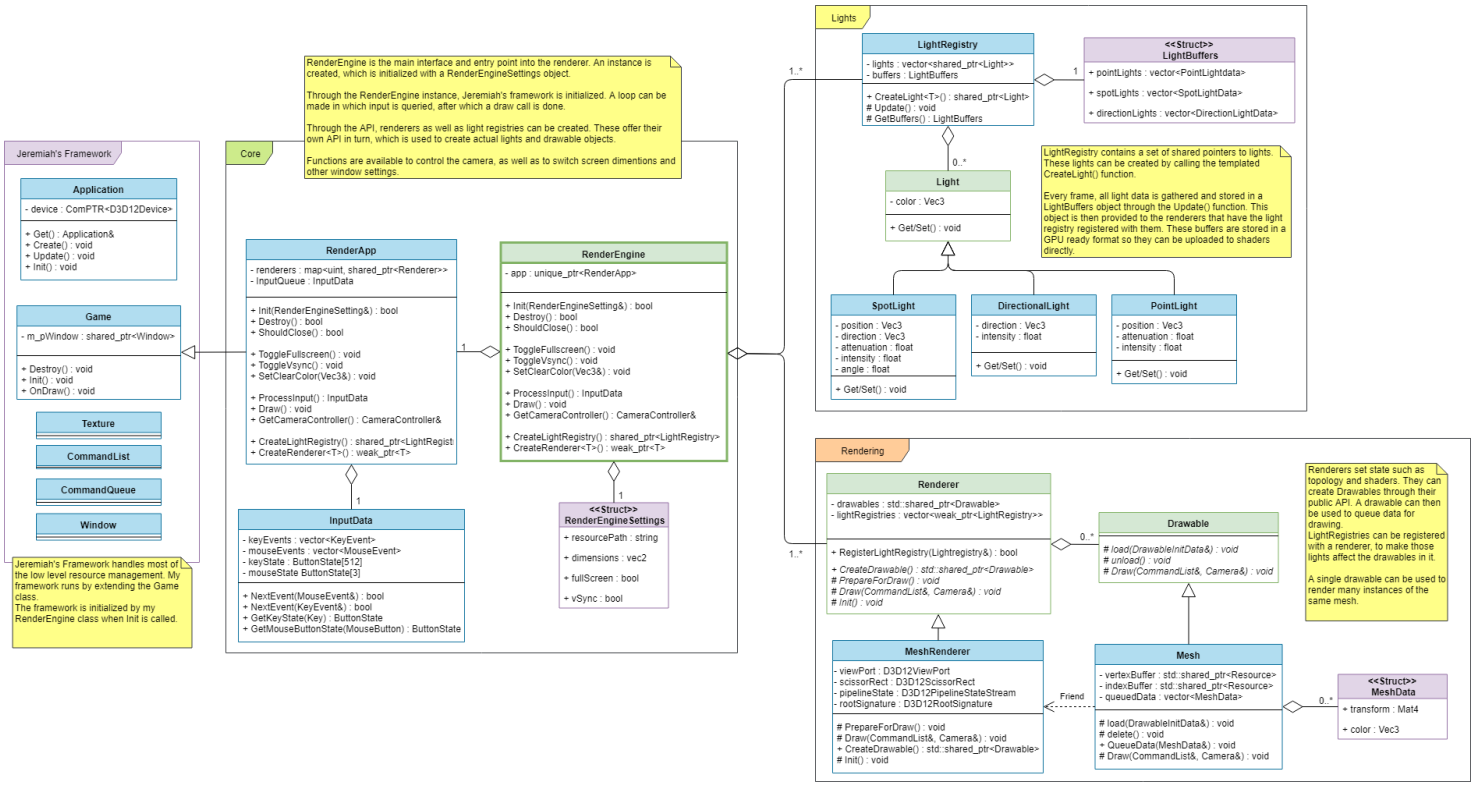
Conclusion

At first I will implement very basic forward shading, simply because it is easier to get setup initially. Once I have that in place, I will look into getting a deferred renderer set up. The game has very few transparent surfaces and many many lights. This means that there's a lot to be gained from going this direction. For the few transparent surfaces that are present (some particles and water in rare occasions), an extra render pass is required + light calculations.

In the articles linked above, efficient techniques through the use of spheres for applying these lights can be found.

Implementation

Interface:



Further details:

Due to time limitations, only a forward shading model has been implemented. Through the use of the above API, meshes, sprites, UI and lines can be drawn. These all follow the format as the Mesh drawable.

Renderer Structure

The renderer should work together with the engine closely, but should not entangle too much.

Requirements

- *Provide an API to the engine to be able to request rendering of models and effects.*
- *Expose only the data that needs to be exposed.*

Solution(s)

A Separate Render Engine Project

This approach will mean the entire render engine will be written in a separate C++ project. It will be compiled as a library that can be used by the engine.

Advantages:

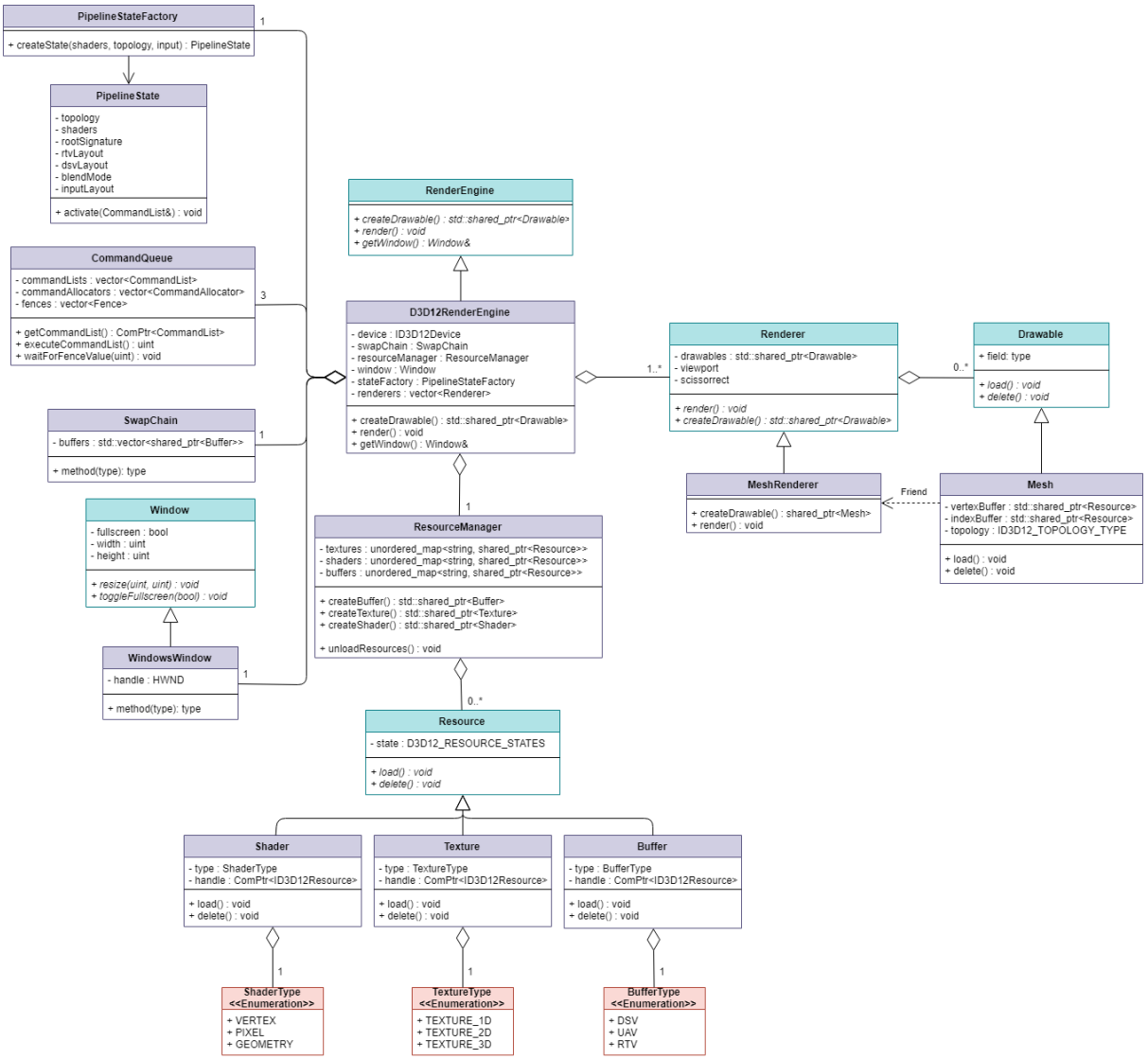
- *All render code is abstracted away, providing a simple interface for the engine programmer to use.*
- *All responsibilities are separated. Render code will not clutter the engine, meaning it will be easier to maintain.*
- *An abstract API means that porting to another graphics API in the future will be easier.*
- *Perforce will not limit who can edit a file at a given time, since the files are strictly separated.*
- *An API can be created early on, with temporary implementations to allow the team to continue. A final efficient implementation can then be worked on without blocking the rest of the team from progressing.*

Disadvantages:

- *It can be more difficult to account for all possible scenarios that the engine programmer might need later on.*
- *It can be difficult to get the right data in the right place in an efficient manner.*

Conclusion

We have opted to put all rendering related code in a separate project, and then expose an API that can be used to create drawables and mark objects for drawing with the data that is required. In the UML below you can see the general layout of the rendering engine project. Each type of drawable has its own render logic.


PipelineStateFactory
+ createState(shaders, topology, input) : PipelineState
PipelineState
- topology
- shaders
- rootSignature
- rtvLayout
- dsvLayout
- blendMode
- inputLayout
+ activate(CommandList&) : void
CommandQueue
- commandLists : vector<CommandList>
- commandAllocators : vector<CommandAllocator>
- fences : vector<Fence>
+ getCommandList() : ComPtr<CommandList>
+ executeCommandList() : uint
+ waitForFenceValue(uint) : void
SwapChain
- buffers : std::vector<shared_ptr<Buffer>>
+ method(type): type
Window
- fullscreen : bool
- width : uint
- height : uint
+ resize(uint, uint) : void
+ toggleFullscreen(bool) : void
WindowsWindow
- handle : HWND
+ method(type): type
RenderEngine
+ createDrawable() : std::shared_ptr<Drawable>
+ render() : void
+ getWindow() : Window&
D3D12RenderEngine
- device : ID3D12Device
- swapChain : SwapChain
- resourceManager : ResourceManager
- window : Window
- stateFactory : PipelineStateFactory
- renderers : vector<Renderer>
+ createDrawable() : std::shared_ptr<Drawable>
+ render() : void
+ getWindow() : Window&
ResourceManager
- textures : unordered_map<string, shared_ptr<Resource>>
- shaders : unordered_map<string, shared_ptr<Resource>>
- buffers : unordered_map<string, shared_ptr<Resource>>
+ createBuffer() : std::shared_ptr<Buffer>
+ createTexture() : std::shared_ptr<Texture>
+ createShader() : std::shared_ptr<Shader>
+ unloadResources() : void
Renderer
- drawables : std::shared_ptr<Drawable>
- viewport
- scissorrect
+ render() : void
+ createDrawable() : std::shared_ptr<Drawable>
Drawable
+ field: type
+ load() : void
+ delete() : void
MeshRenderer
+ createDrawable() : shared_ptr<Mesh>
+ render() : void
Friend
Mesh
- vertexBuffer : std::shared_ptr<Resource>
- indexBuffer : std::shared_ptr<Resource>
- topology : ID3D12_TOPOLOGY_TYPE
+ load() : void
+ delete() : void
Resource
- state : D3D12_RESOURCE_STATES
+ load() : void
+ delete() : void
Shader
- type : ShaderType
- handle : ComPtr<ID3D12Resource>
+ load() : void
+ delete() : void
Texture
- type : TextureType
- handle : ComPtr<ID3D12Resource>
+ load() : void
+ delete() : void
Buffer
- type : BufferType
- handle : ComPtr<ID3D12Resource>
+ load() : void
+ delete() : void
ShaderType
<<Enumeration>>
+ VERTEX
+ PIXEL
+ GEOMETRY
TextureType
<<Enumeration>>
+ TEXTURE_1D
+ TEXTURE_2D
+ TEXTURE_3D
BufferType
<<Enumeration>>
+ DSV
+ UAV
+ RTV
1
3
1
1
1
1..*
0..*
0..*
1
1
1
1

Model Loading

GLTF models need to be loaded from file by the graphics engine.

Requirements

- *Load GLTF models from a file.*
- *Provide the GLTF data in a way that D3D12 can use.*

Solution(s)

Fx-gltf Library

Fx-gltf is a library that can load GLTF files in C++.

Advantages:

- *This library was designed to work with DirectX12, and has many examples of loading and rendering models in C++. After loading a file, the data can be used with D3D12 directly.*
- *The library is contained within a single header file.*
- *The library has been unit tested and used by many people. This makes the chance for bugs minimal.*

Disadvantages:

- *The library depends on another C++ JSON library. Though this is not really a disadvantage, as we'd need a JSON library anyways in order to read GLTF files.*

References/links:

- https://github.com/jessey-git/fx-gltf
- https://github.com/nlohmann/json

Conclusion

Using these libraries will save a lot of time, which can then be spent on making the game look better. Fx-gltf is the go-to library.

Implementation

Interface:

Through the use of the Drawable interface shown in UML's before, GLTF model files can be loaded in. Most specifically through the StaticMesh drawable by providing a file path to the gltf file.

Resource (Texture) Loading and Managing

Resource files need to be loaded to video memory. First the file needs to be stored in RAM, after which the data is uploaded to the GPU. The data needs to be accessible to the renderer and drawable objects.

Requirements

- *Load image files (PNG, JPG, BMP) from file to memory.*
- *Read the data correctly based on how it is stored in the file.*
- *Keep track of loaded resources, and unload them if they aren't being used.*

Solution(s)

STB Image

STB Image is a library that can load image files from file into memory in C and C++.

Advantages:

- *We used this library in previous blocks, so we know how it works.*
- *It is lightweight, and has a single header implementation.*
- *Various formats supported.*

References/links:

- https://github.com/nothings/stb

Jeremiah's Framework

The Framework Jeremiah provides uses DirectXTex to handle texture loading. This is already built in.

Advantages:

- *It is already implemented, so we don't have to worry about it.*
- *It supports every format we require.*

Reference Count

Every time a specific texture is loaded, its reference count goes up. This can be in combination with loading a model or sprite. When the engine then unloads a specific model or sprite, its texture reference count decreases. When it reaches zero, the texture is removed from memory. Accessing a resource would be through the use of a handle.

Advantages:

- *C-like API so it might be easier to port.*

Disadvantages:

- *Resource managing becomes quite tedious when a handle is used. It also makes it harder to use the API for the graphics programmer, as actual data is all hidden.*
- *Looking up the resource belonging to a handle adds CPU cycles.*

References/links:

- https://github.com/jessey-git/fx-gltf/tree/master/examples/viewer
- https://www.3dgep.com/learning-directx-12-4/

Shared Pointer

When a resource is loaded, it is contained by a wrapper object. A shared pointer to this wrapper object is returned. This way multiple meshes can own and use the same texture. This avoids having any lookup times introduced through the handle. A copy of the shared pointer has to be kept in the resource managing class, bundled with its file name when applicable. This makes it possible to copy the shared pointer when a resource would normally be double loaded. This saves GPU memory. It also means that a resource is not released when it is no longer referenced. This is required because D3D12 required manual synchronization. When it is safe to deallocate resources, all resource shared pointers will be iterated over. The ones that no longer have outside references will then be safely unloaded.

Advantages:

- *Makes use of modern C++, and thus offers a much cleaner API.*
- *No lookup times from handles, every resource owning entity can directly access it.*

References/links:

- https://github.com/jessey-git/fx-gltf/tree/master/examples/viewer

Conclusion

We decided to go with Jeremiah's framework and the texture solution it provides. Making all the resource managing software ourselves would be too much work and not doable. Shared pointers do almost all the resource managing for us, and are faster in almost every way.

2D sprites

In R-Type Delta, there are many 2D sprites on the screen. Some are used as animated explosions, others are used in particle systems where there is many of the same 2D sprite. Sprites are used to represent missiles and bullets as well. The sprites exist in 3D space, and are not all at the same depth as the player ship.

Requirements

- *The ability to draw a 2D sprite on a quad facing the camera (billboard).*
- *The ability to efficiently draw thousands at the same time.*
- *Sprites can be transparent, so some require Z-sorting.*
- *Expose an API to the engine that allows the creation of 2D sprites by providing a texture file. The sprite can then be queued for drawing by providing a transform.*

Solution(s)

Instanced Billboards

All sprites will be drawn on a quad that faces the camera. This is cheap, and allows for instancing. That means a single draw call is required to draw every sprite (from missiles to particles). All sprites that are transparent will be handled separately and Z-sorted. Light is then separately applied to them and the layers are finally merged together.

Advantages:

- *Instancing means only a single draw call is required to draw every sprite in the scene.*

Disadvantages:

- *In case of a deferred renderer, an extra step is required for all transparent sprites.*

References/links:

- https://learnopengl.com/Advanced-Lighting/Deferred-Shading
- https://martindevans.me/game-development/2015/10/09/Deferred-Transparency/
- http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/billboards/

Conclusion

By using instanced billboards, the sprites will always face the camera and drawing them is extremely cheap. Transparency will be handled separately in an extra render pass in the deferred renderer.
Light interacts with the particles, which are flat. This means that light may appear a bit odd. I will have to look into ways to mitigate this and make them appear less flat.

Particle Systems

In R-Type Delta, there are many particle systems. They exist in 3D space, and spawn 2D sprites. These sprites have a lifetime, and can fade out over time. They have a velocity, acceleration and direction. This system could be part of the engine programmers responsibilities, but this way calculations can possibly be offloaded to the GPU.

Requirements

- *Particle systems need to create many particles.*
- *Created particles are assigned data based on the ranges defined in the pool, for example a lifetime between a min and max value.*
- *Particles can fade out over time.*
- *Particle systems have a lifetime themselves.*
- *A flexible API needs to be provided to the engine so that any sort of particle effect can be created with the provided parameters.*

Solution(s)

Particle Pool

Particles will be stored in a pool of continuous memory to be able to quickly iterate over them for updating, and to reuse particles that have died.

Advantages:

- *No memory fragmentation.*
- *Less memory allocations.*

Disadvantages:

- *Maximum cap on the amount of particles per pool.*

Depth Buffer Collision

When updating particles, calculations can be done on the GPU to compare them against the depth buffer of the current scene. This gives the illusion that particles can bounce off walls.

Advantages:

- *Realistic particle simulation.*
- *The GPU is great at doing such calculations, saving CPU power.*

Disadvantages:

- *Slightly more complex than not doing it this way.*

Particle System API

An API that allows the creation of particle systems in 3D space. All particle parameters can be specified in ranges, and the size of the particle pool can be specified as well. Provide lambda callbacks which allow particles to be manipulated in a custom way per particle system.

Advantages:

- *Freedom for the engine and gameplay programmers.*
- *No need to expose the messy details of the system.*

Disadvantages:

- *Possibility that some effects are not possible if the API is not designed flexible enough.*

Conclusion

A particle pool + flexible API are surely the way to go. If time permits, offloading calculations to the GPU to allow for terrain collision simulation will be a neat addition.

Gameplay entities

The gameplay we are going to recreate it based on stage 2 of R-Type Delta. Specifically after one minute of gameplay LINK TO VIDEO. The gameplay will only last one minute.

Heres a brief list of everything that will be featured:

Player:

- *The player can move in two axes X and Y.*
- *The player can shoot in front of itself.*
- *The player can die.*
- *The player can pick up upgrades*

Probe:

- *The probe can snap to the front and back of the player.*
- *When disconnected can move on it's own*
- *The probe can fire.*
- *The probe can be upgraded*

Enemies:

- *Can move along a path*
- *Can shoot bullets*
- *Can create other entities*

Powerups:

- *Change the behavior of the probe*

Bullets:

- *Can interact with other entities*
- *Bullets may be affected by gravity*

Lazers:

- *Bounce of terrain*

Rockets:

- *Can track and home on to nearby enemies*

Camera:

- *Will follow a path*

Template

Feature

Describe the feature by answering: What is the feature? And why do we need it?

Requirements

- *What does this feature need to do?*
- *What problems does it solve?*

Solution(s)

Solution 1

Describe the solution. What is this solution?
Diagrams/images, please add them!

Advantages:

- *Why should we pick this solution?*

Disadvantages:

- *What problems has this solution?*
  Potential solution: is there anything we can do against these disadvantages?

References/links:

- Websites, articles, any research to confirm the advantages and disadvantages. Any link to what you're talking about. Libraries.

Conclusion

Which solution or solutions should we pick? Why?

Implementation

Interface:

How do we interact with this feature?
Diagrams/images, please add them!

Further details:

Links to other documents. List of files or classes involved.